# The Polyhedral Model Beyond Loops Recursion Optimization and Parallelization Through Polyhedral Modeling

Salwa Kobeissi, Philippe Clauss

# The Polyhedral Model Beyond Loops

## Recursion Optimization and Parallelization Through Polyhedral Modeling

Salwa Kobeissi
Inria Camus, ICube Lab., CNRS,
University of Strasbourg
Strasbourg, France
salwa.kobeissi@inria.fr

Philippe Clauss
Inria Camus, ICube Lab., CNRS,
University of Strasbourg
Strasbourg, France
philippe.clauss@inria.fr

## Abstract

There may be a huge gap between the statements outlined by programmers in a program source code and instructions that are actually performed by a given processor architecture when running the executable code. This gap is due to the way the input code has been interpreted, translated and transformed by the compiler and the final processor hardware. Thus, there is an opportunity for efficient optimization strategies, that are dedicated to specific control structures and memory access patterns, to apply as soon as the actual runtime behavior has been discovered, even if they could not have been applied on the original source code.

In this paper, we develop this idea by identifying code extracts that behave as polyhedral-compliant loops at runtime, while not having been outlined at all as loops in the original source code. In particular, we are interested in recursive functions whose runtime behavior can be modeled as polyhedral loops. Therefore, the scope of this study exclusively includes recursive functions whose control flow and memory accesses exhibit an affine behavior, which means that there exists a semantically equivalent affine loop nest, candidate for polyhedral optimizations. Accordingly, our approach is based on analyzing early executions of a recursive program using a Nested Loop Recognition (NLR) algorithm, performing the affine loop modeling of the original program runtime behavior, which is then used to generate an equivalent iterative program, finally optimized using the polyhedral compiler Polly. We present some preliminary results showing that this approach brings recursion optimization techniques into a higher level in addition to widening the scope of the polyhedral model to include originally non-loop programs.

***Keywords*** Compilation, dynamic analysis, loop optimization, parallelization, polyhedral model, recursion, recursive function

**ACM Reference Format:**
Salwa Kobeissi and Philippe Clauss. 2018. The Polyhedral Model Beyond Loops: Recursion Optimization and Parallelization Through Polyhedral Modeling. In *Proceedings of 9th International Workshop on Polyhedral Compilation Techniques - in conjunction with HiPEAC 2019 (IMPACT 2019).* ACM, New York, NY, USA, 7 pages.

*IMPACT 2019, January 21-23, 2019, Valencia, Spain*

.

## 1 Introduction

In the area of automatic program optimization and compilation for imperative languages, loop structures are obviously legitimate targets, since they represent a large and compute-intensive part of the whole execution time. Accordingly, numerous advanced loop analysis and optimization techniques have been developed and implemented in compilers and other tools dedicated to optimizing code transformations. Many of these sophisticated loop optimizers are based on the polyhedral model from which they acquire their powerful capabilities in this domain (e.g., Polly [8], Pluto [5]).

The polyhedral model is a mathematical framework that contributes a substantial abstraction and representation for programs having affine loop nests and accessing multi-dimensional arrays through affine array references, i.e., programs with static control parts (SCoPs). This framework provides powerful analysis, and aggressive loop automatic optimizing and parallelizing transformations (e.g., loop tiling, loop skewing, loop interchange, etc. ).

The Apollo framework [14, 21] shows that even if loops that are outlined in a source code do not have the required shape, i.e., are not SCoPs, they may still be good candidates for polyhedral optimizations since their actual runtime behavior is similar to SCoPs' runtime behavior: actual loop bounds and memory references can be modeled as affine functions of surrounding loop indices. In this paper, we go further with this idea of discovering a "SCoP" runtime behavior for code extracts that are even not outlined as loops in the source code, but as recursive functions. Compared to what has been achieved with Apollo, the general goal is not only to build an affine modeling of the data flow, but also to build an affine (loop) modeling of the control flow.

Among the time-expensive structures, recursive functions also play an important role. They generally implement complex algorithms that may scan huge data structures such as graphs, trees and matrices. A recursive approach is usually adopted when solving any problem whose solution relies on smaller and simpler instances of its own. It also eases the expression of computations that are generic to parameters as search depths or problem dimensions.

It is well-known that recursive and iterative (loops) approaches can be used alternately to solve a problem. Replacing a recursive code by an equivalent iterative code and

vice versa is usually possible [2]. In particular, tail recursive functions are automatically handled by production compilers (CLANG, GCC) and transformed into equivalent loops. However, the so-generated loops are generally not suitable for advanced polyhedral optimizations. Moreover, unlike loops, recursive functions do not take advantage of very advanced automatic parallelization and optimization techniques [9, 15].

In this paper, we present our promising idea to make use of the polyhedral model's powerful capabilities to optimize and parallelize recursions, through a guided modeling of their runtime behaviors as polyhedral-compliant loops. Note that the proposed approach is different than static recursion elimination, since it mostly ignores the syntactic shape of the target code. It should rather be understood as a dynamic and automatic rewrite of the target recursive code, through the polyhedral modeling of a part of its data and control flow. Our approach mainly consists of a dynamic analysis technique to check for a linear looping memory and control behavior for a given recursion. Thus, before replacing the original recursive code, we make sure that, at runtime, it acts as an affine for-loop. Subsequently, we generate a semantically equivalent iterative code and, finally, apply further polyhedral optimizations.

The paper is organized as follows : Section 2 discusses some representative previous works on recursion optimizations and parallelizations. Section 3 describes the steps of our method and the tools utilized to implement our idea, and Section 4 supports our proposal by showing some preliminary results. Finally, Section 5 sums up the work and proposes some future works concerning our purpose.

## 2 Related Work

Various studies exist for the purpose of optimizing and parallelizing recursive functions. Nevertheless, the proposed parallelization techniques are mainly static and involve task parallelization where several recursive calls or invocations are run simultaneously.

Rugina and Rinard [18] present a compiler to parallelize divide-and-conquer recursive functions using pointer analysis and symbolic analysis to detect independent recursive calls and generate code that executes these calls concurrently. Also, for divide-and-conquer implementations, Gupta *et al.* [9] propose a compile-time framework that uses interprocedural symbolic array section analysis to detect the independence of multiple recursive calls of divide-and-conquer algorithms. It also proposes a speculative run-time parallelization technique when static analysis is not sufficient. Besides, a tool called Huckleberry [7] automatically parallelizes sequential recursive divide-and-conquer implementations for multi-core platforms.

Another approach [17] proposed by Morihata and Matsuzaki is an automatic parallelization method for recursive functions using quantifier elimination, where the input structure is partitioned into chunks that are run in parallel. In addition, Mizutani et al. [16] propose a different strategy to parallelize recursive functions where programmers decide on using simple or dynamic load balancing depending on the workload of the recursive functions and on the threshold up to which each call is executed in parallel. Saougkos et al. [19] propose an automatic fine-grained parallelism extraction method for recursive functions having integer variables updated in a systematic way.

DECAF [10] is a technique to optimize recursive task parallel programs by reducing the task creation and termination overheads. Tetzlaff and Glesner [23] propose a machine learning based approach to statically predict the recursion frequency of functions used to guide various hot spot optimizations.

Sundararajah *et al.* [22] propose recursion twisting transformations for nested recursions that improve data locality. However, their technique is solely devoted to specific shapes of recursions where data is organized in two trees, called inner and outer trees, and where recursive calls are nested.

A different approach for optimizing recursive implementations is recursion elimination. In this context, the work presented by Bird [3] proposes recursive elimination methods for some forms of recursions using stacks. Cohen [6] presents four types of redundancy in recursive functions and suggests a solution for each of these types by eliminating recursion without using a stack. Another recursion elimination technique is tackled by Bird [4], so-called tabulation, which addresses repetitive (redundant) evaluation of certain values by only computing values for once and, then, storing them for later uses. Also, Liu and Stoller [13] describe a method for transforming some recursive programs into programs using the dynamic programming design technique. Stitt and Villarreal [20] propose recursion flattening that can eliminate many instances of recursion by determining recursion depth, and then inlining recursive calls. However, it cannot eliminate all recursions.

Adriadne [15] is a compiler that extracts directive-based parallelism from recursive function calls. It extracts three forms of parallelisms and a transformation for each one: (1) recursion elimination: converting recursive function to iterative one, (2) parallel-reduction eliminating recursion and distributing workload into independent tasks, (3) thread-safe parallelizing recursive functions that contain independent recursive calls. So, Adriadne for one of its approaches performs recursion-to-iteration transformation, but it only supports recursive functions whose parameters remain unchanged among recursive calls, except for one integer parameter, the so-called index variable, participating in all the conditions and the termination of the recursive function.

Furthermore, some compilers today (e.g. clang/LLVM [12]) implement an optimization pass for transforming recursive functions into loops, however limited to tail recursions, i.e.,

to recursive functions where the recursive call is the last instruction of the function. Additionally, this transformation is not incremental and, thus, is not able to transform several nested recursive calls which would become tail after its application. Other well-known recursion elimination and recursion-to-loop transformation techniques are not yet automated in these compilers.

To sum up, recursive implementations may take advantage of parallelism and other optimizations as described in the studies above. However, the finest parallelism granularity is usually one recursive function invocation where data locality optimization cannot be handled very accurately. When transformations of recursive functions into loops are proposed, the resulting loops are generally too complicated for taking advantage of efficient compilers loop optimizations, thus not providing significant performance improvement regarding their recursive counterpart. This is mostly due to the employed static approaches that try to build a generic equivalent loop program, where either the call stack is simulated using a dedicated data structure, or the generated loops are while-loops with dynamic termination conditions, or the loop bodies embed many conditional branches to mimic the original cases of recursive calls.

In this paper, we show that dynamic analysis of recursive functions and guided modeling of the runtime actual behavior enable the generation of simpler equivalent loops that may be affine, i.e. polyhedral-compliant.

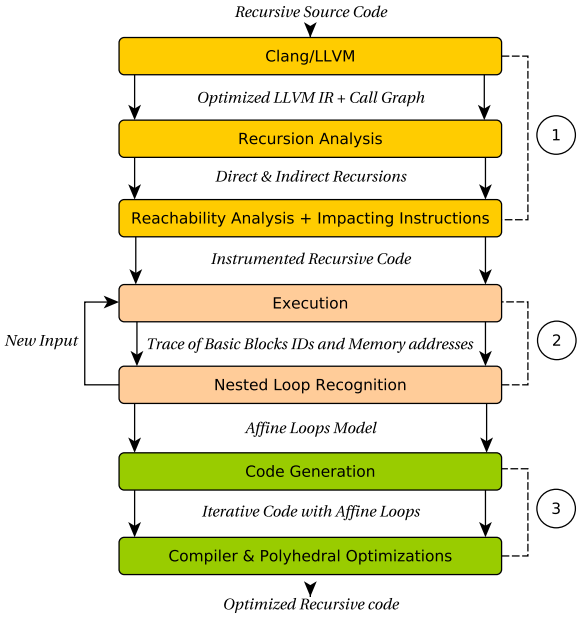


**Figure 1.** General View of the Framework

## 3 From Recursive Functions to Optimized Loops

In this section, we present a prototype of the implementation of our approach as a proof of concept for modeling recursive functions as affine loops and applying polyhedral optimizations.

Our framework involves the steps schematized in Figure 1 which can be summarized as three main phases: (1) Recursive control and memory behavior analysis, (2) Recursion to affine loop modeling and (3) Code generation and polyhedral optimizations. In the input recursive program, we distinguish two kinds of instructions:

1. Instructions whose role is exclusively dedicated to direct the control flow. In the case of recursive functions, such instructions are used to decide about function invocations and the setting of some parameters related to further potential (recursive) calls.
2. Instructions that participate in the computation and memory store of the final result of the program. Such instructions act directly or indirectly for the updating of the data structure that stores the final result. They constitute the minimal set of instructions of the original recursive program that, if conveniently scheduled and instantiated, have to be run to reach the (correct) final result. We call such instructions *impacting instructions*.

Our framework is aimed at handling recursive programs whose control and memory runtime behavior can be characterized as follows:

- Basic blocks of the LLVM intermediate representation that contain impacting instructions are scheduled at runtime as if they would belong to nested loops whose respective unique index has affine loop bounds depending on the surrounding loop indices.
- The target addresses of memory accesses regarding the main input and output data structures can be modeled as affine functions of surrounding loop indices;
- The general affine loop structure modeling the control and memory behavior must be independent of the processed input. By “input-independent”, we mean that the number and depths of the loop nests remains the same regardless of the input, while only some loop bounds and some coefficients of the memory reference functions may change accordingly.

We describe in details the steps of our framework in the following subsections. As illustrating example, we use a recursive implementation of the matrix product shown in Listing 1.

```
1  void MatrixMultiplication(int A[N][N], int B[N][N
       ])
2  {
3    static int row=0, column=0, index=0;
4
5    if (row >= N)
6      return;
7
8    if(column < N){
9      if(index < N){
10       C[row][column]+=A[row][index]*B[index][
       column];
11       index++;
12       MatrixMultiplication(A, B);
13     }
14     index=0;
15     column++;
16     MatrixMultiplication(A, B);
17   }
18   column=0;
19   row++;
20   MatrixMultiplication(A, B);
21 }
```

**Listing 1.** Recursive Matrix Multiplication C Function

### 3.1 Recursive Control and Memory Behavior Analysis

In this first phase, we execute the original recursive program given a relatively small input. If the recursion's behavior is affine, then the analysis result of the execution can be used to build a structure of equivalent loop nests. However, we may need to analyze several executions of the program to predict some loop bounds with respect to the input. The analysis results are then used for extrapolating the program's behavior for larger inputs. This phase involves several steps that are detailed in the following subsections.

#### 3.1.1 Compile-Time Classical Optimizations

First of all, given an input recursive source code, we let the LLVM compiler apply some classical optimizations (e.g., promote memory to register) to prepare the intermediate representation (IR) of our input program for the later steps. We do not activate the tail call elimination pass which transforms tail recursive calls into loops for two reasons: (1) the way the target recursive function is transformed may not result in an affine loop and (2) if there are several recursive calls in the target code, only one tail call may be eliminated.

**Example 3.1.** Consider again the matrix product shown in Listing 1. The recursive function is direct and invokes three recursive calls to itself. Notice that the last recursive call, on line 17, is a tail call. When this function is compiled with the tail call elimination pass activated, the tail call is removed and a loop is created. Nevertheless, there are still two remaining recursive calls, where the second one is now a tail call. However, since the tail call elimination pass is not applied incrementally by the compiler, these remaining calls are not eliminated. Thus, we do not activate this pass in order to let our framework handle all the three recursive calls in the same way.

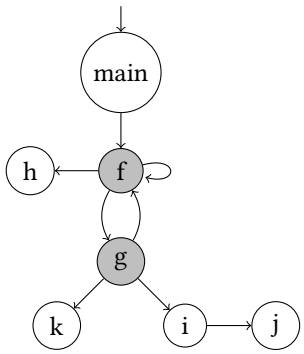


**Figure 2.** Example of a Call Graph of an Arbitrary Recursive Program

#### 3.1.2 Compile-Time Recursion Detection

At this step, our framework checks, at compile time, if the program involves any recursions, and, if so, identifies these recursions and the functions participating in them. In order to detect recursions, it uses the call graph corresponding to the IR of the program. The call graph is a directed graph representing relationships between functions in a program, where a node represents a function and an edge from one node to another represents a call such that the source node constitutes the caller function and the destination is the callee. Figure 2 shows an example of a call graph of a program made up of seven functions: *main*, $f$, $g$, $h$, $i$, $j$, $k$ such that:

- function *main* calls $f$
- $f$ invokes itself, $h$ and $g$
- $g$ calls back $f$, $k$ and $i$
- $i$ calls $j$

In this example, we notice that function $f$ is in a direct recursion with itself and an indirect recursion through function $g$.

After obtaining the call graph, we search for strongly connected components (SCC) in it such that an SCC is a subgraph where every node is reachable from every other node. In this context, a cycle in SCC implies that a recursion exists among the functions represented by the nodes involved in this cycle. If the cycle is made up of only one node (loop), then it is a direct recursion. Else, it is indirect. In our example of Figure 2, we have an SCC involving the two gray nodes of functions $f$ and $g$ such that we have a loop over $f$ signifying a direct recursion and a cycle from $f$ to $g$ and $g$ back to $f$, implying an indirect recursion among $f$ and $g$.

#### 3.1.3 Recursion Reachability Recognition

Eventually, we need to trace the control and memory behavior of recursions and find out if they act as static control parts. Thus, we are interested in tracking basic blocks containing impacting instructions, whether they are executed directly or indirectly by the recursive functions. For this reason, in addition to the recursive functions themselves, our tool also determines their reachability in the program. By reachability, we mean all the functions that can be reached by a sequence of calls initiated by the recursive functions themselves. In Figure 2, the reachability of the recursive functions $f$ and $g$ includes:

- $h$ (directly called by $f$)
- $k$ and $i$ (directly called by $g$)
- $j$ (indirectly called by $g$ through $i$)

**Example 3.2.** In the matrix product example, function multiplyMatrix is the only recursive function, and it does not call any other function than itself.

#### 3.1.4 Impacting Instructions

We identify impacting instructions in the following way. In the IR of the program, our framework marks the writes to the data structure which is the final output of the recursion and their corresponding reachable functions. Then, for every write instruction of this kind, it marks all instructions leading and contributing to it, i.e., their backward static slices (BSS). A backward static slice, is a set of instructions existing in the code of a program that may affect a certain value, i.e., in our case, a value of the data structure which is the final output of the recursion. Slicing the important parts of a given recursive program (i.e., recursive functions and their reachability functions) is essential for the analysis phase of our implementation since it helps keeping track of the elemental parts of recursions that must be taken into consideration for further transformations and optimizations.

**Example 3.3.** In the LLVM IR of the recursive matrix product, the memory writes related to matrix $C$ in function multiplyMatrix, as well as their backward static slices are marked. These backward static slices include the memory reads related to matrices $A$, $B$ and $C$, and the computations related to the updates of $i$, $j$ and $k$, since they are involved in the computations of the memory addresses referenced by the memory instructions.

#### 3.1.5 Profiling

Finally, printing statements are added to the IR to collect the IDs of executed basic blocks that include impacting instructions, and that would be enough to run to reach the final result if they are well initiated. Other blocks that are only dedicated to the recursive control are not instrumented. Additionally, writes to memory and reads from memory are to be instrumented to collect the corresponding target memory addresses, with the final purpose of building affine reference functions.

The program executes for a first time, and the output expected is the memory and control trace of recursions in the program. Subsequently, we investigate if this trace can be modeled as affine loop nests. For this purpose, we use the Nested Loop Recognition (NLR) algorithm originally presented in [11].

#### 3.1.6 Nested Loop Recognition (NLR) Algorithm

The NLR algorithm takes as input a trace of a program execution and constructs a sequence of loop nests that produce the same original trace when run. The applications of this algorithm include: (1) program behavior modeling for any measured quantity such as memory accesses, (2) execution trace compressing and (3) value prediction, i.e, extrapolating loops under construction (while reading input) to predict incoming values.

In our tool, not only do we use NLR to model memory accesses, which is one of its original goals, but also to model sequences of basic blocks IDs, which is more singular. Given our trace of a target recursive program, generated thanks to our instrumentation, if NLR builds affine loop nests including the interesting basic blocks IDs and memory addresses interpolated by the constructed loop indices, then the generation of equivalent affine loops may be performed.

An initial NLR model is built from running the instrumented code with small data input. Then, additional runs with different data inputs are performed and their associated NLR models are compared to the initial one, in order to find out if (1) the general affine loop structures are maintained across the models and (2) if the differences between the models can be identified as being solely loop bounds or coefficients of memory reference functions. Then, their respective values are interpolated relatively to some input parameters, as typically the problem size. Constant parts of the affine functions modeling memory references may obviously vary across the models, since they mostly represent base addresses of data structures.

**Example 3.4.** Listing 2 offers a shortened view of the result of NLR. The runtime behavior of function multiplyMatrix is modeled by NLR as two affine loop nests including memory accesses that are expressed as affine combinations of the loop indices. The first loop nest is obviously dedicated to the main computations of the matrix product, while the second loop nest models the returns of the recursive calls. Note that loop bounds and some memory reference coefficients are set to their generic value related to the problem size $N$, thanks to polynomial interpolation resulting from comparing the modeling of several runs with different problem sizes. Items MEM1, MEM2 and MEM3 represent the base addresses of matrices $A$, $B$ and $C$ respectively.

```
for i0 = 0 to N-1
  for i1 = 0 to N-1
    for i2 = 0 to N-1
      val MatrixMultiplication::if.then4 //IR basic block
      ...
      load // memory read
      val MEM1 + 4*N*i0 + 4*i2 //memory address in terms of loops indices
      ... //repetitive memory access patterns
      load
      val MEM2 + 4*i1 + 4*N*i2 //4 is the size of an integer
      ...
      val load
      val MEM3 + 4*N*i0 + 4*i1
      val store // memory write
      val MEM3 + 4*N*i0 + 4*i1
      ...
    val MatrixMultiplication::if.end15
    ...
  val MatrixMultiplication::if.end17
  ...
for i0 = 0 to N*N-1
  for i1 = 0 to N-1
    val MatrixMultiplication::if.end17
    ...
    val MatrixMultiplication::if.end15
    ...
  val MatrixMultiplication::if.end15
  ...
```

**Listing 2.** Recursive Matrix Multiplication NLR model

### 3.2 Recursion to Affine Loop Transformation

This phase requires both the NLR model and the original recursive program itself. Code generation and transformation are achieved as follows. According to the loop nests provided by NLR, our framework extracts their structures and generates corresponding loop nests in LLVM form using the suitable basic blocks, taking into consideration changing the accessed memory addresses to be in terms of the loop indices as it is in the NLR model. Then, they are inserted into the LLVM IR of the program replacing the targeted recursions.

### 3.3 Polyhedral Optimizations Application

The final resulting affine loop nests replacing the original recursive functions may already take advantage of standard compiler loop optimizations (loop invariant code motion, constant propagation, common subexpression elimination, etc.). However, they are also potential candidates for polyhedral optimizing and parallelizing transformations. For this purpose, our framework applies Polly [1] on the LLVM IR of the affine loop nests.

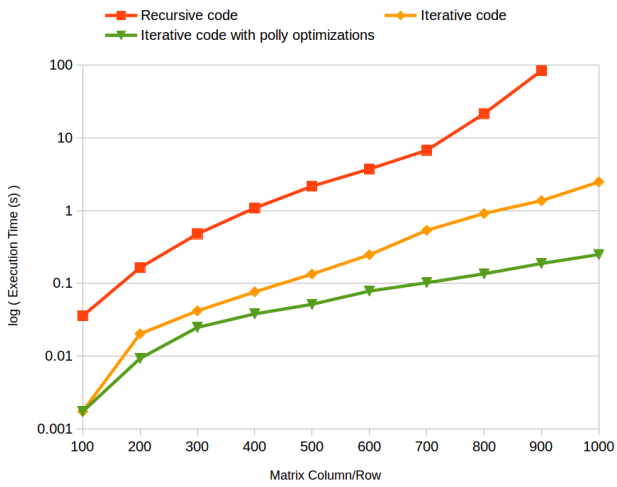


**Figure 3.** Performance results with the matrix product

## 4 Experiments

In this section, we show the results of our experiments by applying the described approach to the matrix product that was used as illustration all along the paper. The computer used is an Intel(R) Core(TM) i7-7Y75 CPU @ 1.30GHz running Ubuntu 16.04.5 LTS. We use Polly with llvm/clang version 8.0.0.

We compare the execution of the original recursive code against the iterative code, either handled solely by clang with flag -O3 or also handled by Polly. Our measures are represented by the curves in Figure 3 using a logarithmic scale for the execution times, and by varying the matrix sizes from 100 to 1,000. It clearly shows that the iterative code outperforms the recursive one, and particularly when polyhedral optimizations have been applied by Polly.

## 5 Conclusion and Perspectives

We have shown that recursive functions may act at runtime as affine loops, and hence be also good candidates for polyhedral optimizations. A dedicated automatic recursion-to-affine-loop transformation made of static and dynamic analysis and transformation passes could be applied whenever possible, on any program embedding recursive functions. Such an approach brings handled recursive functions to a higher level of optimizations, compared to what has been achieved until now regarding recursive functions in compiler research. It also extends the polyhedral model applicability to non-loop control structures.

Yet, our work is still a proof of concept for this approach, and we aim at investigating more in this promising idea to implement an efficient specialized compiler for this sake covering many more and complicated recursive programs. Some of the future works, that should be done in order to achieve this purpose, include:

1. Performing dynamic analysis for recursive behavior at runtime on-the-fly, instead of analyzing an early program execution with a smaller input. This is because some recursive functions might not completely behave as affine loop nests. Instead they might involve non continuous large static control parts during their execution. Thus, at runtime, we keep checking for linear behavior and, accordingly, transform the code for executing these certain parts.
2. Inducing verification features to the framework so dynamic analysis can be possible and the iterative model constructed can be considered predictive.
3. Tackling input dependent recursive codes.

## References


[1] Polly - llvm framework for high-level loop and data-locality optimizations. https://polly.llvm.org/index.html.

[2] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Trans. Program. Lang. Syst.*, 4(2):295–322, Apr. 1982.

[3] R. S. Bird. Notes on recursion elimination. *Commun. ACM*, 20(6):434–439, June 1977.

[4] R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, Dec. 1980.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008.

[6] N. H. Cohen. Characterization and elimination of redundancy in recursive programs. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 143–157, New York, NY, USA, 1979. ACM.

[7] R. L. Collins, B. Vellore, and L. P. Carloni. Recursion-driven parallel code generation for multi-core platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 190–195, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[8] T. Grosser, A. Größlinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[9] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, Dec 2000.

[10] S. Gupta, R. Shrivastava, and V. K. Nandivada. Optimizing recursive task parallel programs. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 11:1–11:11, New York, NY, USA, 2017. ACM.

[11] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 94–103, New York, NY, USA, 2008. ACM.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[13] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1):37–62, Mar 2003.

[14] J. M. Martinez Caamano, M. Selva, P. Clauss, A. Baloian, and W. Wolff. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience*, 29(15), June 2017.

[15] A. Mastoras and G. Manis. Ariadne - directive-based parallelism extraction from recursive functions. *J. Parallel Distrib. Comput.*, 86(C):16–28, Dec. 2015.

[16] Y. Mizutani, D. Nakajima, N. Fujimoto, and K. Hagihara. Evaluation of a compiler with user-selectable execution strategies for parallel recursion. *Systems and Computers in Japan*, 35(9):92–103.

[17] A. Morihata and K. Matsuzaki. Automatic parallelization of recursive functions using quantifier elimination. In *Proceedings of the 10th International Conference on Functional and Logic Programming*, FLOPS'10, pages 321–336, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. *SIGPLAN Not.*, 34(8):72–83, May 1999.

[19] D. Saougkos, A. Mastoras, and G. Manis. Fine grained parallelism in recursive function calls. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 121–130, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[20] G. Stitt and J. Villarreal. Recursion flattening. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 131–134, New York, NY, USA, 2008. ACM.

[21] A. Sukumaran-Rajam and P. Clauss. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim.*, 12(4):48:1–48:27, Dec. 2015.

[22] K. Sundararajah, L. Sakka, and M. Kulkarni. Locality transformations for nested recursive iteration spaces. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 281–295, New York, NY, USA, 2017. ACM.

[23] D. Tetzlaff and S. Glesner. Static prediction of recursion frequency using machine learning to enable hot spot optimizations. In *IEEE 10th Symposium on Embedded Systems for Real-time Multimedia, ESTIMedia 2012, Tampere, Finland, October 11-12, 2012*, pages 42–51, 2012.